A Novel Incremental Approach to Association Rules Mining in Inductive Databases*

Rosa Meo, Marco Botta, Roberto Esposito, and Arianna Gallo

Dipartimento di Informatica, Università di Torino, Italy
{meo, botta, esposito, gallo}@di.unito.it

Abstract. Constraints-based mining languages are widely exploited to enhance the KDD process. In this paper we propose a novel *incremental* approach to extract itemsets and association rules from large databases. Here incremental is used to emphasize that the mining engine does not start from scratch. Instead, it exploits the result set of previously executed queries in order to simplify the mining process. Incremental algorithms show several beneficial features. First of all they exploit previous results in the pruning of the itemset lattice. Second, they are able to exploit the mining constraints of the current query in order to prune the search space even more. In this paper we propose two incremental algorithms that are able to deal with two — recently identified — types of constraints, namely item dependent and context dependent ones. Moreover, we describe an algorithm that can be used to extract association rules from scratch in presence of context dependent constraints.

1 Introduction

The problem of mining association rules and, more generally, that of extracting frequent sets from large databases has been widely investigated in the last decade [1,2,3,4,5,6]. These researches addressed two major issues: on one hand, performance and efficiency of the extraction algorithms; on the other hand, the exploitation of user preferences about the patterns to be extracted, expressed in terms of constraints. Constraints are widely exploited also in data mining languages, such as in [5,7,8,9,10,11] where the user specifies in each data mining query, not only the constraints that the items must satisfy, but also different criteria to create groups of tuples from which itemsets will be extracted. Constraint-based mining languages are also the main key factor of inductive databases [12], proposed in order to leverage decision support systems. In inductive databases, the user explores the domain of a mining problem submitting to the system many mining queries in sequence, in which subsequent queries are very often a refinement of previous ones. This might be, if not properly addressed, a huge computational workload. This problem becomes even more severe considering that these queries are typically instances of iceberg queries [13], well-known to be expensive on very large databases such as common data warehouses. In such systems the intelligent exploitation of user constraints becomes the key factor for a successful exploration of the problem search space [14]. The same occurs also in "dense" datasets, in which the volume of

* This work has been funded by EU FET project cInQ (IST-2000-26469).

J.-F. Boulicaut et al. (Eds.): Constraint-Based Mining, LNAI 3848, pp. 267–294, 2005.

the result set (the frequent itemsets) compared to the input data is particularly large. In these cases, constraints are exploited to make a problem tractable that otherwise would require a non affordable computational workload [15].

In this application context, in order to speed up the query execution time, it makes sense to exploit the effort already done by the DBMS with previous queries. In fact, inductive databases can materialize the result of (some of the) previous queries. In this way, previous results are available to the mining engine which can "reuse" some of the information contained in them in order to reduce the workload. Indeed, since nowadays the storage space is critic to a lesser extent, our aim is to reduce as much as possible the computational work of the data mining engine. Furthermore, we suppose that the mining engine works in an environment similar to a data warehouse, in which database content updates occur rarely and in known periods of time. This greatly simplifies the task, since previous results are considered up to date and can be usefully exploited to speed up the execution of current queries. Therefore, we suggest that the execution plan of a constraint-based query should take into consideration also the results of previous queries, already executed and readily available. The necessity of storing and exploitiong a collection of query results, has been recognized previously also in [16] in which they propose a rule query language for the postprocessing of rules based on their statistical properties or elements.

We present here an incremental approach (originally proposed in [17]) that computes the result of a query starting from the result of a previous, more general query. The new result is computed by enforcing on the previous result set the new constraints. We notice that several "incremental" algorithms have been developed in the data mining area [18,19,20,21], but they address a different issue: how to efficiently revise the result set of a mining query when the database source relations get updated with new data. In this Chapter we show that the new incremental query evaluation technique is beneficial and reduces the system response time. First of all because previous results allow pruning of the itemset lattice. Secondly, because the mining constraints of the current query allow to prune the search space even more. Of course, we assume that the system relies on an optimizer who is entitled to recognize query equivalence and query containment relationships in order to identify the most convenient result from which starting the incremental computation. [22] describes a prototype of such an optimizer and shows that its execution time is negligible (in the order of milliseconds) for most practical applications.

Our main contributions here, are two incremental algorithms that provide a fast solution to the case of query containment. The first one exploits the somewhat implicit assumption made in almost all previous works in constraint-based mining: properties on which users define constraints are functionally dependent on the item to be extracted, i.e., the property is either always true or always false for all occurrences of a certain item in the database. In this case, it is possible to establish the satisfaction of the constraint considering only the properties of the item itself, that is, separately from the context of the database in which the item is found (e.g., the purchase transaction). In [22], we characterized the constraints that are functionally dependent on the item extracted and called them *item dependent* (ID) constraints. The exploitation of these constraints proves to be extremely useful for incremental algorithms. Indeed, ID constraints allow the selection

of the valid itemsets in advance, based on their characteristics, that hold separately from the transactions in which itemsets are found. Similar reasoning occurs also for *succinct* constraints [23] with the difference that these latter consist in properties to be evaluated on each single item separately from the other items of the itemset. In other words, they do not foresee aggregate properties on itemsets.

In [22], another class of constraints, namely the *context dependent* (CD) constraints, was introduced as well. Context dependent constraints occur very often in many important application domains, such as in business (e.g., analysis of stock market data) [24] in science (e.g., meteorological forecast) [25] but also in the traditional applications of data mining, such as in market basket analysis [26].

In order to offer a first grip to the intuition about this novel concept, let us explain it by means of a simple example which applies to the analysis of stock market data. To this purpose, we assume that the database to be analyzed contains the attributes:

`date` — the date of interest;
`stock` — the name of the stock;
`price` — a categorical attribute assuming values in {`increased`, `not varied`, `decreased`}.

In such a context, the user may be interested in whether there exists any (negative) correlation between groups of stock items. For instance, it may want to associate stocks for which `price=increased` with ones for which `price=decreased` instead. As an example of the result, he/she may find useful to discover that $\ll$*when price of AT&T and Microsoft stocks increase, then the price of Sun Microsystems decreases with a probability of* 78% $\gg$.

In this case, the stock price, which is the feature on which constraints are evaluated, does not depend only on the stock, but it also depends on another variable (time). Time and stock together provide the context in which price is determined. Therefore, in contrast to ID constraints, the satisfaction of CD constraints cannot be decided without reading the contextual information present in the database transaction. While, in the simplest situations, the problem may be solved by filtering the database relation before the mining process, such a filtering is not possible when different predicates are given for the body part and the head part of the rules or when constraints on aggregates must hold on the sets. In this latter cases, CD constraints proved to be very difficult to be dealt with. In fact, a CD constraint is not necessarily satisfied by a certain itemset in all its instances in the database. And this fact has big influence on the possibilities of pruning that constraints allow on the lattice search space. In fact, even if an itemset, satisfying a CD constraint *within a transaction*, satisfies one of the well studied properties of monotonicity or anti-monotonicity over the itemset lattice, the same properties do not necessarily hold for that itemset in the whole database. Unfortunately, most of the state of the art algorithms [4,23,27], are based instead on the principle that those properties hold for a certain pattern database wide.

As far as incremental mining is concerned, the presence of a CD constraint in a query implies that one needs to carefully check whether the constraints are satisfied by scanning the transaction table. In the following, we present a new algorithm which is able to deal with context dependent constraints. We show that incremental algorithms are valuable tools even in this setting.

Despite the lack of studies on algorithms dealing with contextual characteristics of itemsets, CD constraints have revealed to be of a certain importance in the extraction of knowledge from databases. For instance, in [26] the authors claim it is important to identify the contextual circumstances in which patterns hold. They propose to reason on circumstances organizing them in a lattice and searching there the most general circumstances in which patterns hold. Their work, however, restricts reasoning on circumstances to conjunctive statements and makes no use of available query results. On the contrary, CD constraints proposed here can be organized with no restriction and queries are allowed to contain general boolean predicates. Secondly, the proposed approach makes a significant usage of available results of previous queries (if the incremental approach results effective with respect to a conventional execution, i.e., by scratch). The incremental option for a data mining algorithm is of course preferable in an inductive database system, since it allows the exploitation of all the available informations in the system in order to speed up the response time.

The rest of the paper is organized as follows. Section 2 presents some preliminary definitions, discusses the properties of queries of the containment relationship. Section 3 and Section 4 present two incremental algorithms able to deal with item dependent and context dependent constraints respectively. Section 5 shows a first set of experimental assessments of the incremental algorithms. In order to fully evaluate the validity of the incremental algorithms in general and with a fair comparison, since in literature there are no algorithms that extract association rules with context dependent constraints, we propose in Section 6 a baseline miner algorithm (called CARE) that is non-incremental and is able to extract association rules with item or context dependent constraints. Finally, in Section 7 we study the worthiness of the incremental approach using the CARE algorithm as a baseline miner. Section 8 draws some conclusions.

2 Preliminary Definitions and Notation

Let us consider a database instance D and let T be a database relation having the schema TS=$\{A_1, A_2, \ldots, A_n\}$. A given set of functional dependencies Σ over the attribute domains $dom(A_i)$, $i = 1..n$ is assumed to be known.

As a running example, let us consider a fixed instance of a market basket analysis application in which T is a `Purchase` relation that contains data about customer purchases. In this context, TS is given by {`tr, date, customer, product, category, brand, price, qty`}, where: `tr` is the purchase transaction identifier, `customer` is the customer identifier, `date` is the date in which the purchase transaction occurred, `product` is the purchased product identifier, `category` is the category to which the product belongs, `brand` is the manufacturer of the product, `price` is the product price, and `qty` is the quantity purchased in transaction `tr`. The Σ relation is {`product` $\rightarrow$ `price`, `product` $\rightarrow$ `category`, `product` $\rightarrow$ `brand`, {`tr, product`} $\rightarrow$ `qty`, `tr` $\rightarrow$ `date`, `tr` $\rightarrow$ `customer`}. It should be noted, however, that the validity of the framework is general, and that it does not depend on either the mining query language or the running database example.

Of course, the above schema could also be represented over a set of relations and dimensions adopting the usual data warehouse star schema. However, the non-normalized

form is more readily explained as well as more common in a data mining environment. In fact, data mining practitioners usually preprocess the data warehouse in order to obtain a database schema similar to the one introduced.

As above mentioned, the system tries to exploit past results in order to react more promptly to user requests. In order to work, such a system must be able to recognize that syntactically different queries are, actually, similar. The following definition, which introduces the notion of *grouping equivalence*, allows the system to recognize that two ways of partitioning the database are equivalent. Clearly, this could be done by building the partitions and checking whether they are identical. However, this approach is clearly too costly. Instead, our approach is to exploit known domain knowledge in order to obtain an answer without actually accessing the database.

Definition 1. *Grouping equivalence relationship.*

Two sets of attributes K_1 and K_2 are said to be `grouping equivalent` *if and only if for any relation T defined on TS:*

$$\forall t_1, t_2 \in T : t_1[K_1] = t_2[K_1] \Leftrightarrow t_1[K_2] = t_2[K_2]$$

where $t_1[K_1]$ is the projection of the tuple t_1 on the attributes in K_1.

Put in other words: K_1 and K_2 are grouping equivalent if and only if $K_1 \leftrightarrow K_2$ (where $\leftrightarrow$ denotes a bidirectional functional dependence).

Example 1. *As we pointed out, the grouping equivalence relation has been introduced as a tool to distinguish whether two set of attributes partition the database in the same way. The following table reports a case where the set of attributes* {`tr`} *and* {`date`, `customer`} *are grouping equivalent, while, for instance* {`date`} *and* {`product`} *are not.*

tr	data	customer	product
1	10/1/2005	CustomerA	Milk
1	10/1/2005	CustomerA	Bread
1	10/1/2005	CustomerA	Beer
2	10/1/2005	CustomerB	Meat
2	10/1/2005	CustomerB	Biscuits
3	12/1/2005	CustomerA	Milk
3	12/1/2005	CustomerA	Biscuits

Let us notice, however, that this example is an oversimplification of what expressed in Definition 1. In fact, here it is reported only a single database instance and we asked to check the grouping equivalence relationship on it. The definition, instead, requires the relation to hold for all *database instances of a given database schema. This actually implies that the relation must be checked using known functional dependencies.*

Sets of attributes that are grouping equivalent form a grouping equivalence class E. As an example of the usefulness of this concept, let us notice that if two queries differ only for the way the relations are grouped and the grouping attributes used in

the two queries are in the same equivalence class, then the two queries are bound to be equivalent.

We assume to know about a set of grouping equivalence classes $E_1 \ldots E_j$.

Example 2. *In the* `Purchase` *example, the following non trivial equivalence class may be found:*

E_1= {{`tr`}, {`date, customer`}, {`tr, date`}, {`tr, customer`}, {`tr, date, customer`}}.

In writing a mining query, the user must specify, among the others, the following parameters:

- The *item attributes*, a set of attributes whose values constitute an item, i.e., an element of an itemset. The language allows one to specify possibly different sets of attributes, one for the antecedent of association rules (body), and one for the consequent (head).
- The *grouping attributes* needed in order to decide how tuples are grouped for the formation of each itemset. The grouping attributes, for the sake of generality and expressiveness of the language, can be decided differently in each query according to the purposes of the analysis.
- The *mining constraints* specify how to decide whether an association rule meets the user needs. In general, a mining constraint takes the form of a boolean predicate which refers to elements of the body or of the head (possibly on the values of any of the attributes in TS, e.g., kind of product, price or quantity). Since we want to allow different constraints on the body and on the head of the association rules, we admit two separate constraint expressions for each part of the rule.
- An expression over a number of *statistical measures* used to reduce the size of the result set and to increase the relevance of the results. This evaluation measures are evaluated only on the occurrences of the itemsets that satisfy the mining constraints.

Usually in market basket analysis, when the user/analyst wants to describe by means of itemsets the most frequent sales occurred in purchase transactions, the grouping attribute is `tr` (the transaction identifier) and the itemsets are formed by the projection on `product` of sets of tuples selected from one group. However, for the sake of generality and of the expressive power of the mining language, grouping can be decided differently in each query. For instance, if the analyst wants to study the buying behavior of customers, grouping can be done using the `customer` attribute, or if the user wants to study the sales behaviour over time he/she can group by `date` or by week or month in the case these attributes were defined.

Users may exploit the mining constraints in order to discard uninteresting itemsets and to improve the performances of the mining algorithm.

More formally, a mining query for the extraction of association rules is described as the 7-tuple:

$$Q = (T, G, I_B, I_H, \Gamma_B, \Gamma_H, \Xi)$$

where: T is the database table; G is the set of grouping attributes; I_B and I_H are the set of item attributes respectively for the body and the head of association rules; Γ_B and

Γ_H are boolean expressions of atomic predicates specifying constraints for the body and for the head of association rules; Ξ is an expression on some statistical measures used for the evaluation of each rule.

We define an atomic predicate to be an expression in the form:

$$\mathtt{A_i}\theta\mathtt{v_{Ai}}$$

where θ is a relational operator such as $<, \leq, =, >, \geq, \neq$, and $\mathtt{v_{Ai}}$ is a value from the domain of attribute A_i.

Ξ is defined to be a conjunction in which each term has the form

$$\xi\theta\mathtt{v}$$

where ξ is a statistical measure for the itemset evaluation, v is a real value, and θ is defined as above.

For the sake of simplicity, in this paper we focus on the *support count* and *confidence* statistical measures. The extension to other measures should be straightforward. The support count is the number of distinct groups containing both the itemsets in the association rule. Confidence is the ratio between the association rule support and support of the body.

Example 3. *The query*

$$Q=(\texttt{Purchase}, \{\texttt{tr}\}, \{\texttt{product}\}, \{\texttt{product}\},\ \texttt{price}>100, \texttt{price}\geq 200,\ \textit{support count}\geq 20 \textit{ AND confidence}\geq 0.5)$$

over the Purchase *relation (first parameter) extracts rules formed by* products *in the body (third parameter) associated to* products *in the head (fourth parameter), where all the products in the rule have been sold in the same transaction (second parameter). Moreover, the* price *of each product in the body must be greater than 100 (fifth parameter) and the* price *of each product in the head must be greater or equal to 200 (sixth parameter). Finally, the support count of the returned rules must be at least 20 and the confidence of the rules at least 0.5. Even if the query syntax we gave is best suited for the purposes of this paper, it is quite unfriendly when it comes to understandability. Many languages have been proposed in the literature that can easily express the kind of constraints we introduced in this paper. For instance, query Q could be expressed in the* Minerule *[8] language as follows:*

MINERULE Q
SELECT DISTINCT $1..n$ $\texttt{product}$ AS BODY, $1..n$ $\texttt{product}$ AS HEAD, SUPPORT, CONFIDENCE
WHERE BODY.$\texttt{price}> 100$ AND HEAD.$\texttt{price}\geq 200$
FROM $\texttt{Purchase}$
GROUP BY $\texttt{tr}$
EXTRACTING RULES WITH SUPPORT COUNT:20, CONFIDENCE: 0.5

Now that we have seen how constraint-based mining queries are formed, let us define two particular types of constraints: the *item dependent* constraints and the *context dependent* ones. In the following, we will denote by $X \rightarrow Y$ a functional dependency (FD) between two attribute sets X (LHS) and Y (RHS) in the database schema TS.

Definition 2. *Dependency set.*
A dependency set *of a set of attributes X contains all the possible RHS that can be obtained from X following a FD $X \rightarrow Y$ in Σ (direct or transitive) such that there is no $X' \subset X$ such that $X' \rightarrow Y$.*

As we did for equivalence classes, we assume to know about a set of dependency sets.

Example 4. *The dependency set of* {`product`} *is* {`category, price, brand`}.*The dependency set of* {`tr`} *is* {`customer, date`} *while the dependency set of* {`tr, product`} *is* {`qty`}. *As a consequence, one can safely assume that:*

- *the value of* `product` *can be used to determine the values of attributes* `category, price`, *and* `brand`;
- *the value of* `tr` *and* `product` *uniquely determines the value of* `qty`.

Definition 3. *Context dependent and item dependent constraints.*
Given a query

$$Q = (T, G, I_B, I_H, \Gamma_B, \Gamma_H, \Xi)$$

let us consider an atomic predicate $\mathtt{P(A)} \in \Gamma_B$ *(respectively* Γ_H*).* $\mathtt{P(A)}$ *is defined to be an* item dependent constraint *if and only if* `A` *belongs to the dependency set of* I'*, where* $I' \subseteq I_B$ *(respectively,* $I' \subseteq I_H$*). If* $\mathtt{P(A)}$ *is not an item dependent constraint, then it is a* context dependent constraint.
A query Q is said to be item dependent if all atomic predicates in Γ_B and Γ_H are item dependent constraints. If Q is not item dependent, then it is context dependent.

We notice that an itemset $\mathcal{I}$ satisfies an item dependent constraint either in any database group in which it occurs, or in none. This immediately implies the following:

Lemma 1. *Statistics for itemsets with item dependent constraints.*
An itemset $\mathcal{I}$ that satisfies an item dependent constraint in a mining query has a statistical measure that is a function of the total number of groups in which $\mathcal{I}$ occurs in the given database instance.

On the contrary, a context dependent constraint might possibly be satisfied by some, but not all, occurrences of itemset $\mathcal{I}$. Then, the statistical measure cannot be evaluated on the number of groups in which the itemset appears. In fact, this number may differ from (i.e., be larger than) the number of groups in which the itemset satisfies the constraint.

We now give some properties that allow to identify the existence of the containment relationship between two mining queries. These conditions provide the theoretical ground supporting the algorithms that we discuss in Section 3 and 4. Through all the discussion we will denote with Q the query issued by the user at the present time and that we want to "optimize", and with Q^i the ones in the past queries repository.

To begin with, we notice that not all Q^i are suitable candidates for testing the containment relationship. In fact, many of them may be built using features which immediately hinder the possibility of finding the relationship to hold. In the following, we

present a simple test which can be used to filter out those queries very efficiently. Further work is probably needed under this viewpoint, but for the moment we consider this aspect as future work. For the time being, we define the concept of candidate queries for containment of Q (that is those queries that may contain Q) as follows.

Definition 4. *Candidates for query containment of Q.*
Let us consider

$$Q^i = (T, G^i, I_B^i, I_H^i, \Gamma_B^i, \Gamma_H^i, \Xi^i)$$
$$Q = (T, G, I_B, I_H, \Gamma_B, \Gamma_H, \Xi)$$

A previous query Q^i is a candidate for containment of Q, if the following conditions hold:

1. *G^i is in the same grouping equivalence class of G*
2. *I_B^i is in the same grouping equivalence class of I_B*
3. *I_H^i is in the same grouping equivalence class of I_H*

Therefore, a first criterion for selecting the candidate queries for containment of Q can be based on testing the above three conditions. That is, we require that the queries are "grouping equivalent" (i.e., both the queries partition the input data in the same groups) and that they use an "equivalent" description for the items in the body and in the head part of the association rules.

Let us denote by R the set of association rules returned by Q and by R^i the set of association rules returned by Q^i.

Theorem 1. *Properties of Query Containment.*
Given

$$Q^i = (T, G^i, I_B^i, I_H^i, \Gamma_B^i, \Gamma_H^i, \Xi^i)$$
$$Q = (T, G, I_B, I_H, \Gamma_B, \Gamma_H, \Xi)$$

Let Q^i be a candidate for containment of Q.
Let the following hypothesis of entailment between constraints of Q and Q^i be fulfilled:

$$\Gamma_B \Rightarrow \Gamma_B^i$$
$$\Gamma_H \Rightarrow \Gamma_H^i$$
$$\Xi \Rightarrow \Xi^i$$

Under these conditions, $R \subseteq R^i$. Furthermore, the support count (`sup_count`*) of an association rule $r \in R$ is upper bounded by the support count (*`sup_count`^{i}*) of the same rule in R^i.*

Proof. Assuming that Q^i is a candidate for containment of Q implies that Q and Q^i partition the database in the same groups and extract from them the same sets of potential association rules. That is, if the two queries did not contain any constraint, then their result sets would be identical. Let us call the result set of the unconstrained query

R'. In order to prove the containment relation, it is left to show that any rule r which is selected from R' by Q is also selected by Q^i. This is actually immediate, in fact, for any such rule: $r \in R \Rightarrow \Gamma_B(r) \wedge \Gamma_H(r) \wedge \Xi(r) \Rightarrow \Gamma_B^i(r) \wedge \Gamma_H^i(r) \wedge \Xi^i(r) \Rightarrow r \in R'$. Here, the first and the last implications hold since, by definition, a rule belongs to the result set of a query Q' if and only if it belongs to the unconstrained version of Q' and satisfies all the constraints in it. The middle entailment, instead, is directly implied by the assumptions of the theorems.

Moreover, any itemset which satisfies Γ_B in a database group also satisfies Γ_B^i in the same group. The same holds for the head constraints. In addition, there might exist some database groups in which Γ_B^i and Γ_H^i are satisfied, but Γ_B or Γ_H are not. Therefore, the support count of any rule r in R^i is bound to be not lower than the support count of r in R.

The following lemma specializes the previous theorem for queries with item dependent constraints.

Corollary 1. *Query containment with item dependent constraints.*

Under the same hypotheses of Theorem 1, let us also assume that the queries are item dependent. Then, a rule $r \in R \cap R^i$ has the same support count in both Q and Q^i. A rule r, such that $r \in R^i$ but $r \notin R$, has a support count equal to zero in Q.

Proof. By definition, an item dependent constraint is satisfied by all the occurrences of a given itemset in the database or by none. Thus, an association rule in R that satisfies Γ_B and Γ_H satisfies also Γ_B^i and Γ_H^i in the same number of groups (and thus satisfies both Ξ and Ξ^i). Therefore its support count will be the same in both the result sets.

However, a rule in R^i which does not satisfy Γ_B and Γ_H will not satisfy them in all the database groups in which it occurs. Thus, it will have a support count equal to zero in Q.

3 An Incremental Algorithm for Item Dependent Constraints

In a previous work [22], we showed that item dependent constraints are particularly desirable from the viewpoint of the optimization of languages for data mining. In particular, we showed that we can obtain the result of a newly posed query Q by means of set operations (unions and intersections) on the results of previously executed queries. We qualify this approach to itemset mining as *incremental* because instead of computing the itemsets from scratch it starts from a set of previous results. In this paper, we are interested in studying the situation of query containment, that is, to consider situations in which query Q imposes a more restrictive set of constraints with respect to a previous query, here denoted with Q^i. In this paper, we show that item dependent constraints can also be exploited to simplify the problem of incremental mining. In fact, it turns out that, in order to retrieve the desired rules, it suffices to identify the rules in the previous results that satisfy the new constraints. As the results in Section 2 imply, this is not generally true in a situation involving context-dependent constraints. In fact, in the latter case, one needs to carefully update the statistical measures of the rules as well (see Section 4).

In Section 2, we showed that under the item dependency assumption, whenever a query Q^i is found to contain Q, it is rather easy to extract the new results from past ones. In fact, it suffices to search in R^i those rules which satisfy the requirements of Q and to copy them verbatim (along with their support counts) into the new result set.

A sketch of this incremental approach is reported in Algorithm 1. The algorithm is very simple: it checks which of the rules in R^i satisfy the constraints in Q and updates R accordingly. It is important to notice that testing Γ_B and Γ_H is a feasible and efficient operation. In fact, since the constraints are item dependent, their evaluation does not require to access the whole (possibly huge) facts table. On the contrary, it merely requires to access the dimension tables and to check the constraints using the informations found therein. Since those tables does usually fit into the main memory or in the DBMS buffer memory, this rarely becomes a demanding operation. In addition, the Ξ constraint is also easily checked by using the statistical measures stored together with the rules in the past result.

Algorithm 1: Item Dependent (ID) incremental algorithm

Data : $R^i = \{b \rightarrow h\}$: old result set;
$Q = (T, G, I_B, I_H, \Gamma_B, \Gamma_H, \Xi)$: the query issued by the user;
Result : R : the set containing the rules satisfying Q
$R \leftarrow \emptyset$;
foreach $r \in R^i$ **do**
  if $\Gamma_B(r) \wedge \Gamma_H(r) \wedge \Xi(r)$ **then**
    $R \leftarrow R \cup \{r\}$;
  end
end

4 An Incremental Algorithm for Context Dependent Constraints

In this section we propose an incremental algorithm which is able to construct the result of a new mining query Q starting from a previous result R^i even when the mining constraints are not item dependent. At the best of our knowledge this is the first attempt to write a mining algorithm able to deal with context dependent constraints [3,4,11,28,29,30].

The algorithm is best described by considering two separate steps. In the first one, the algorithm reads rules from R^i and builds a data structure which keeps track of them. We call this structure the *BHF* (Body-Head Forest) and describe it in Section 4.1. We notice that since the BHF is built starting from a previous result set and represent only rules found therein, this corresponds to a first pruning of the search space. In fact, subsequent operations will simply disregard rules that do not appear in it (the correctness of this approach is implied by Theorem 1).

In the second step, the algorithm considers two relations $T_b = \{< i, g > | i \in I_B, g \in G, \Gamma_B \; is \; true\}$ and $T_h = \{< i, g > | i \in I_H, g \in G, \Gamma_H \; is \; true\}$, containing the items and the group identifiers (GIDs) that satisfy the mining constraints in query Q. T_b and T_h are obtained by evaluating the constraints on the fact table. Their role is

to keep track of the context in which the itemsets appear. In fact, the context dependent constraints require that their validity is checked group by group. The two relations fullfil this purpose. We notice that this is another point in which the search space is pruned. In fact, the constraints are evaluated on the database and the items which do not satisfy the mining constraints are removed, once and for all, from the input relations.

Finally, the algorithm updates the counters in the BHF data structure accordingly to the itemsets found in T_b and T_h. The counters are then used to evaluate the statistical measures needed to evaluate whether the constraint Ξ is satisfied.

4.1 Description and Construction of the BHF

A BHF is a forest containing a distinguished tree (the body tree) and a number of ancillary trees (head trees). The body tree is intended to summarize the itemsets which will be found in the body part of the rules. Importantly, in any tree, an itemset B is represented as a single path and vice versa. In the node corresponding to the end of a path, it is stored a head tree and the (body) support counter.

Analogously, the head tree (associated to the itemset B) is intended to summarize the itemsets that will appear in the head part of the rules (having the body equal to B). A head tree is similar in structure to a body tree with the exception that there are no head trees associated to the end of any path. A path in a head tree corresponds to an itemset H and is associated to a counter which stores the support of the rule.

Figure 1 gives a schematic representation of a BHF.

In the following, we will make use of the following notation: given a node n belonging to a body tree or to a head tree, we denote with n.child(i) the body (respectively the head) tree rooted in the node n in correspondence of the item i. For instance, in the root node of the BHF reported in Figure 1, there are four items, and three non-empty children; root.child(a) denotes the body node containing the items c, d, and z. In a similar way we denote the head tree corresponding to a particular item i in a node n using the notation n.head(i). We also assume that both body elements and head elements are sorted in an unspecified but fixed order. We denote with $B[k]$ (respectively with $H[k]$) the k-th element of the body B (respectively head H) w.r.t. this ordering. Finally, in many places we adopt the standard notation used for sets in order to deal with BHF nodes. For instance, we write $i \in n$ in order to specify that item i is present in node n; we write $n \cup i$ in order to denote the node obtained from n by the addition of item i.

Procedure insertRule

Data : root : the BHF root node
$B \rightarrow H$: the rule to be inserted

headTree $\leftarrow$ insertBody(root, B, 1) ;
insertHead(headTree, H, 1);

Procedure insertRule describes how a rule is inserted in the BHF structure. The procedure consists in two steps. In the first one, the body B of the rule is inserted in the body tree (see Function insertBody) and the node n corresponding the end of the path

associated to B is determined. In the second one, the head is inserted and attached to n (see Procedure insertHead).

We notice that the hierarchical structure of the BHF describes a compressed version of a rule set. In fact, two rules $B_1 \rightarrow H_1$ and $B_2 \rightarrow H_2$ share a sub path in the body tree provided that B_1 and B_2 have a common prefix. Analogously they share a sub path in a head tree provided that $B_1 \equiv B_2$ and H_1 and H_2 have a common prefix.

Function insertBody

Data : n : a BHF node
B : an itemset
k : an integer

if $B[k] \notin n$ **then**
$n \leftarrow n \cup B[k]$
end
if $k < \text{size}(B)$ **then**
insertBody(n.child($B[k]$), $B, k+1$)
else
return n.head($B[k]$)
end

Procedure insertHead

Data : n : a BHF node
H : an itemset
k : an integer

if $H[k] \notin n$ **then**
$n \leftarrow n \cup H[k]$
end
if $k < \text{size}(H)$ **then**
insertHead(n.child($H[k]$), $H, k+1$)
end

4.2 Description of the Incremental Algorithm

Here, we assume that a BHF has been initialized using the rules in the previous result set R^i (but with their support count equal to zero: it will adjusted in the following step). We will show how the BHF is updated and the rules are extracted in order to build the novel result set R.

In the following we will denote with:

- $T_b^{ITEM}[g] \equiv \{i \mid (g,i) \in T_b\}$ and with $T_h^{ITEM}[g] \equiv \{i \mid (g,i) \in T_h\}$ the set of items in group g that satisfy the body and the head constraints, respectively.
- $T_b^{GID} \equiv \{g \mid (g,i) \in T_b\}$ and with $T_h^{GID} \equiv \{g \mid (g,i) \in T_h\}$ the set of GIDs in T_b and in T_h, respectively.
- $T_b^{GID}[i] \equiv \{g \mid (g,i) \in T_b\}$ and with $T_h^{GID}[i] \equiv \{g \mid (g,i) \in T_h\}$ the set of group identifiers in which item i satisfies the body and the head constraints, respectively.

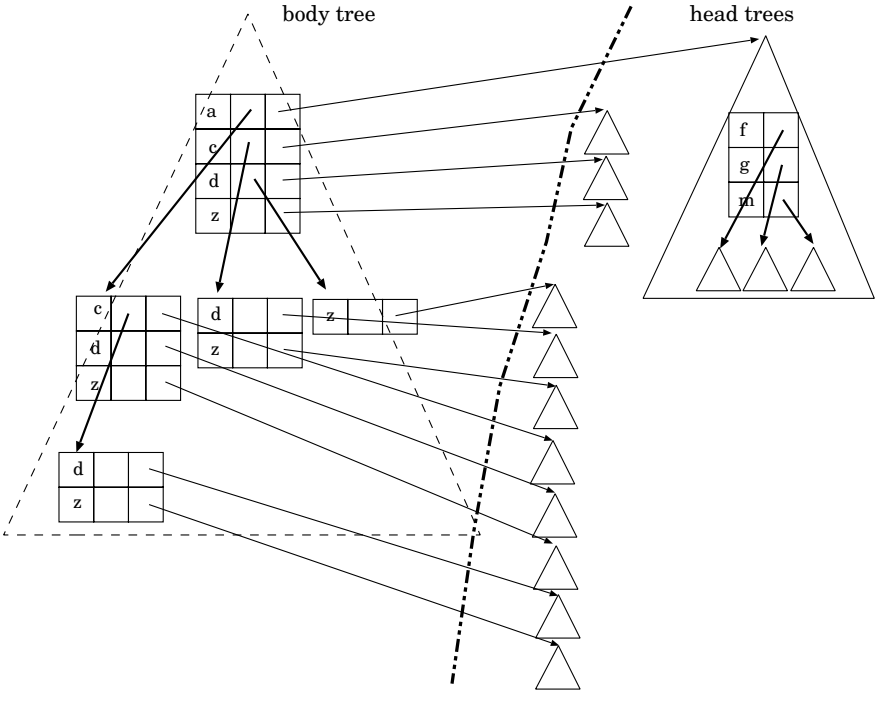


Fig. 1. Example of BHF

- τ the support threshold chosen by the user
- $r.body$ the body of rule r and with $r.head$ the head of rule r

For the sake of readability, we reported in Algorithm 5 a simplified version of the incremental algorithm which has the advantage of making its intended behavior clear. We believe it is self explanatory. Instead, the implemented version greatly improves on the simple, reported one. Let us now assume that the working of Algorithm 5 is clear. We now try to give an idea of how the implemented version works and improve on it. The main difference is that in order to avoid the checking of each and every rules in the BHF (see Procedure incrRuleSupp), the algorithm performs a depth first search in the BHF. In this way it is able to find all the rules which need their support to be updated for a given group g without actually enumerate all possible rules.

Let us illustrate the way the implemented algorithm proceeds by means of an example. Let us assume that the BHF in Figure 1 is given, and for group g it holds $T_b^{ITEM}[g] = \{a, c, z\}$ and $T_h^{ITEM}[g] = \{f, l\}$. In order to update the support counters in the BHF tree, the algorithm proceeds as follows. The root of the body tree is examined in order to check which of the items it contains are in $\{a, c, z\}$. The item "a" is found and its support counter is therefore incremented (we recall that the supports of the body part of the rules are needed in order to evaluate the rules confidence values). Since $T_h^{ITEM}[g] \neq \emptyset$, the head tree associated to "a" is examined. As a result, the algorithm increments the support counters associated to the items in $T_h^{ITEM}[g]$. That is

the ones corresponding to the rules having "a" in their body and any subset of $\{f, l\}$ in their head.

Once the updating of the counters in the head tree rooted in "a" is complete, the algorithm examines the rest of the body tree. In the root node of the sub tree rooted in "a", the algorithm searches whether it contains items belonging to $\{c, z\}$. It finds item "c" and increments its support. As in the previous case, the algorithm examines the head tree associated to this item and updates the support counters accordingly.

Then, the sub tree rooted in "c" is examined in a similar way. Whenever a body tree node does not contain any items in $T_b^{ITEM}[g]$, the algorithm backtracks to its ancestor looking for items in $T_b^{ITEM}[g]$ that have not been "visited" yet in that node.

Algorithm 5: Context Dependent (CD) incremental algorithm

```
Data   : T_b, T_h
Result : R_2
for all GID g ∈ T_b^GID do
    incrRuleSupp(BHF, T_b^ITEM[g], T_h^ITEM[g] )
end
for all rule r ∈ BHF do
    if Ξ(r) then
        R_2 ← R_2 ∪ r
    end
end
```

5 Results

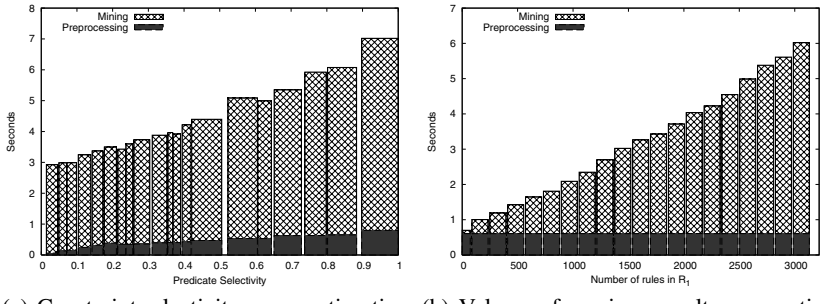
The two incremental algorithms presented in this paper have been assessed on a database instance, describing retail data, generated semi-automatically. We generated a first approximation of the fact table (`purchases`) using the synthetic data generation program described in [31]. The program has been run using parameters $|T| = 25$, $|I| = 10$, $N = 1000$, $|\mathcal{D}| = 10,000$, i.e., the average transaction size is 25, the average

Procedure `incrRuleSupp`

```
Data   : a BHF,
         T_b, T_h
Result : It updates the support counters in the BHF
for all r ∈ BHF do
    if r.body⊆ T_b^ITEM[g] then
        r.body.support++;
        if r.head⊆ T_h^ITEM[g] then
            r.support++;
        end
    end
end
```

size of potentially large itemsets is 10, the number of distinct items is 1000 and the total number of transactions is 10.000. Then, we updated this initial table by adding some attributes which provide the details (and the contextual information) of each purchase. We added some item dependent features (such as "category of product" and "price") and some context dependent features (such as "discount" and "quantity"). The values of the additional attributes have been generated randomly using uniform distributions on the respective domains[1].

We note here how a single fact table suffices for the objectives of our experimentation. While, in fact, the characteristics of the database instance (e.g., total database volume and data distribution) are determinant in order to study the behavior of mining algorithms, this is not so when we are up to study incremental algorithms. Indeed, as simple complexity considerations point out, the important parameters from the viewpoint of the performance study of incremental algorithms are the selectivity of the mining constraints (which determine the volume of data to be processed from the given database instance) and the size of the previous result set.

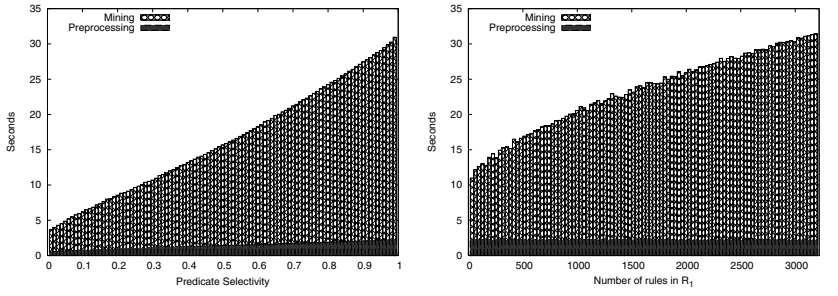


(a) Constraint selectivity vs execution time (b) Volume of previous result vs execution time

Fig. 2. Empirical evaluation of the item dependent (ID) incremental algorithm

In Figure 2(a) we report the performances of the item dependent incremental algorithm (ID) as the selectivity of the mining constraints changes. We experimented different constraints on the item dependent attributes, letting the constraints selectivity vary from 0% to 100% of the total number of items. In Figure 2(a) we sampled twenty points. Figure 2(b) tests the same algorithm, but it lets vary the number of rules in the previous result set. Again we sampled twenty points (in the range $0 \ldots 3220$). The two figures report the total amount of time needed by the algorithm to complete. In particular, the bars, which represent the single experiments, are divided in two components: the preprocessing time (spent in querying the database to retrieve and store in main memory the items that satisfy the constraints), and the core mining time (needed by the algorithm to read the previous result set and to filter out those rules that do not satisfy the constraints any more).

[1] The dataset can be downloaded from `www.cinq-project.org`



(a) Constraint selectivity vs execution time (b) Volume of the previous result vs execution time

Fig. 3. Empirical evaluation of the CD incremental algorithm

Figures 3(a) and 3(b) report the performances of the context dependent (CD) algorithm. The figures report again the total execution time, specifying how much time was spent for preprocessing and for the core mining task. It is worth noticing, that the CD incremental algorithm performs a greater amount of work with respect to the ID algorithm because the problem it solves is far more complex. In fact, in the preprocessing phase the algorithm must retrieve all the group/item pairs satisfying the constraints and access to them in order to build and update the BHF data structure. Only then, it can retrieve the results from the BHF structure.

A couple of points are worth noting. The execution times of both algorithms increase almost linearly with the increase of the two parameters (constraint selectivity and previous results), but, as it was expected, the item dependent incremental algorithm runs much faster than its counterpart.

Moreover, as the experiments in Section 7 will show, both the algorithms are faster than CARE, a new algorithm (that we introduce in the following section) which, nevertheless, is capable of solving a class of more difficult problems - mining association rules in presence of context-dependent constraints.

CARE, as the most of the algorithms, operates starting from scratch. We emphasize here, that, at the best of our knowledge, CARE is the only mining algorithm capable of dealing with context dependent constraints. Hence, comparisons with other mining algorithms on the field of context dependent constraints are difficult to be made. Our guess is that any state of the art mining algorithm should be able to outperform CARE, on problems defined in terms of item dependent constraints, due to the lower generality of the problem they face.

Interestingly, nevertheless, thanks to the good performances of incremental algorithms also on problems with context dependent constraints, one has always the choice of avoiding the use of CARE. This is done by running first the mining algorithm of his choice (on the problem defined by the query but *without* the context dependent constraints) and then applying the incremental algorithm on top of it (with the addition of context dependent constraints). The issue of which choice is the most promising is out of the scope of this paper. However, we believe that the answer is likely to depend on the problem at hand. In particular, whenever the mining constraints select a very small

part of the original dataset, CARE is likely to be very fast. On the contrary, whenever the result set is small, the incremental algorithm seems more efficient.

6 The Constrained Association Rules Extractor Algorithm

CARE (Constrained Association Rules Extractor) is an algorithm which has been designed to extract association rules in presence of context dependent constraints. Context dependent constraints do not allow the classical two phases algorithms (first, find frequent itemsets, second, extract association rules), because frequency count and constraint satisfaction interact. In principle, a levelwise algorithm, such as Apriori [31], might be used, provided that at each iteration the constraint is checked on the database. We decided to develop CARE starting from Partition [32], since it has been shown to perform better than Apriori on many databases.

Even if CARE can work with item dependent constraints, its main purpose is to provide a first and simple solution to the problem posed by CD constraints, thus providing a baseline comparison for incremental algorithms. However, it is necessary to notice that CARE is still under development in order to overcome its limitations: it is not able to deal with cross mining constraints, such as `BODY.feature1 < HEAD.feature1`, or aggregate constraints, such as `sum(BODY.quantity) > val`. Its main features are the following:

- in contrast with any published algorithm we are aware of, CARE uses two separate structures for storing items to be put in the body and items to be put in the head of an association rule. This is needed because constraints on the *body items* might be different from constraints on the *head items*.
- It maintains for each item a list of group identifiers (*gidlists*) of the transactions in which the item appears, as done, for instance, in Partition [32]. This allows, on one hand, to scan the database only twice, and on the other hand, to keep all needed information to combine body items and head items. In a future implementation, on dense databases, bitmaps will be used to efficiently store and operate on gidlists.

In a constrained mining framework, items must be frequent and satisfy the additional mining constraints. In the general settings we are considering, items in the body of a rule should satisfy a mining constraint Γ_B, whereas items in the head of a rule should satisfy a possibly different mining constraint Γ_H. Such constraints can be evaluated a posteriori, i.e., at the time rules are extracted from frequent itemsets, only if the constraints are item dependent.

In the general case of context dependent constraints (e.g., `BODY.qty > 2` $\wedge$ `HEAD.qty = 1`), items might satisfy the constraints Γ_B or Γ_H in some transactions and not in others, thus influencing the frequency with which an item that satisfies the mining constraints occurs. Moreover, as already said, since the same item in a transaction might satisfy only one of the two constraints Γ_B and Γ_H, it is necessary to maintain separate structures for storing *body itemsets* and *head itemsets*. In particular, there is a structure (named BH) for the items satisfying the constraints on the body. Each entry of this structure contains a triple $< i, \mathrm{Bgids}, \mathrm{HH} >$, where i is an item, Bgids is a list of group identifiers (gidlists) in which the item is found in the source table, and HH is a

structure containing the items satisfying the constraints on the head. In this way, every item that can be in the body of a rule is associated with the set of items that can be put in the head of such a rule. This structure is very similar to the BHF previously described, as it only contains the root nodes of the forest. In the following, we will use set notation for the abstract description of the algorithms, while in the actual implementation of these structures we used hashmaps, for efficiency reasons.

Algorithm 7: The CARE algorithm

Data Structures:
Data : T_b, T_h;
ϵ minimum support threshold;
Result : R
BH = $\{ < i, \text{Bgids}, \text{HH} > | i \in I_B,$
$\text{Bgids} = T_b^{GID}[i], |\text{Bgids}| \geq \epsilon,$
$\text{HH} = \{< j, \text{Hgids} > | j \in I_H,$
$\text{Hgids} = \text{Bgids} \cap T_h^{GID}[j], |\text{Hgids}| \geq \epsilon\}$
$\}$
$R = \text{buildRules}(\text{BH}, \epsilon)$;
return R;

CARE works in two steps: in the first step, BH is initialized by scanning the tables T_b and T_h. Then, the rules are extracted through a recursive process.

The full sketch of CARE is reported in Algorithm 7.

Rules are extracted from the BH structure by first creating a body itemset (see function buildRules that in turn calls buildBody), then creating the corresponding head itemsets for that body (see function buildHead), and repeating the process recursively for all items in the BH structure. Note that each itemset in the body (head) is obtained by union of the current body (head) with another item in the BH structure (HH structure). The gidlist of the candidate itemset is obtained by intersection of the respective gidlists of the two "ancestor" itemsets and its cardinality is finally tested to verify the support constraint.

Function buildRules

Data : BH the BH structure ;
ϵ minimum support threshold;
Result : R
while $(\text{BH} \neq \emptyset)$ **do**
  let $e =< i, \text{Bgids}, \text{HH} > \in \text{BH}$;
  $\text{BH} = \text{BH} - \{e\}$;
  $R = R \cup \text{buildBody}((\{i\}, \text{Bgids}), \text{BH}, \text{HH}, \epsilon)$;
end
return R;

Function buildBody

Data : (CurrentBodyItemset, CurrentGids) current body information ;
BH the body-head structure ;
ϵ minimum support threshold;
Result : R the rules extracted
$R = \text{buildHead}(\text{CurrentBodyItemset}, \text{HH}, (\emptyset, \text{CurrentGids}), \epsilon)$;
while $(\text{BH} \neq \emptyset)$ **do**
  let $e =< i, \text{Bgids}, \text{HH} >\in \text{BH}$;
  $\text{BH} = \text{BH} - \{e\}$;
  $\text{newgids} = \text{Bgids} \cap \text{CurrentGids}$;
  if $|\text{newgids}| \geq \epsilon$ **then**
    $\text{newBody} = \text{CurrentBodyItemset} \cup \{i\}$;
    $R = R \cup \text{buildHead}(\text{newBody}, \text{HH}, (\emptyset, \text{newgids}), \epsilon)$;
    $R = R \cup \text{buildBody}((\text{newBody}, \text{newgids}), \text{BH}, \epsilon)$;
  end
end
return R;

Function buildHead

Data : CurrentBodyItemset current body items ;
HH the Head structure ;
(CurrentHeadItemset, CurrentGids) current head information ;
ϵ minimum support threshold;
Result : R the rules extracted
$R = \emptyset$;
while $(\text{HH} \neq \emptyset)$ **do**
  let $e =< i, \text{Hgids} >\in \text{HH}$;
  $\text{HH} = \text{HH} - \{e\}$;
  $\text{newgids} = \text{Hgids} \cap \text{CurrentGids}$;
  if $|\text{newgids}| \geq \epsilon$ **then**
    $\text{newHead} = \text{CurrentHeadItemset} \cup \{i\}$;
    $R = R \cup \{\text{CurrentBodyItemset} \rightarrow \text{newHead}\}$;
    $R = R \cup \text{buildHead}(\text{CurrentBodyItemset}, \text{HH}, (\text{newHead}, \text{newgids}), \epsilon)$;
  end
end
return R;

Let us illustrate the process through a simple example. Suppose we are given the source table reported in Table 1 and that we want to extract rules that satisfy the following constraints:

- `BODY.qty`$\geq$`1 ∧ HEAD.qty`$\geq$`5`
- minimum support count = 2

After reading the source table filtered by *body constraints* and *head constraints*, CARE builds the BH structure as reported in Table 2.

Table 1. Example of a source table

gid	item	qty
1	A	2
	B	7
	C	8
2	A	1
	C	6
3	A	1
	B	6
	C	1

Table 2. The BH structure built from the source table in Figure 1

body item	gidlist	HH structure	
A	1,2,3	head item	gidlist
		B	1,3
		C	1,2
C	1,2,3	head item	gidlist
		B	1,3

It should be noted that there is no entry for body item B, because, even though item B is frequent, there are no frequent items for the head associated to B, i.e., no rules with B in the body can be extracted.

Afterwards, buildRules is called and rules are extracted in the following order:

- $A \rightarrow B$
- $A \rightarrow C$
- $AC \rightarrow B$
- $C \rightarrow B$

i.e., for every body, buildHead is called to build the corresponding heads. Then, buildBody is called recursively to build larger and larger bodies. Finally, the next item in the BH structure is taken into consideration and the process repeated.

Of course, a number of optimizations might be implemented by accurately computing gidlist intersections and storing intermediate results. However, the current implementation is sufficiently efficient to be used for comparison with the incremental algorithms presented in the following sections.

7 Comparison Between the CD Incremental Algorithm and CARE

In this section we compare the performances of the CD incremental algorithm with CARE. In the experiments, we want to observe how the dimensions of the problem impact on the performances of the two algorithms. Thus, we designed the experiments by varying one dimension at a time, so that the influence of each dimension is observed

separately with respect to the other ones. The dimensions are (as already pointed out in Section 5): the selectivity of the mining predicates, the support threshold, and the volume (number of rules) of the previous result set. For each experiment, we report the running time of the two algorithms. We notice that, in general, the problem parameters have a different impact on the two algorithms. For instance, the support threshold is probably the parameter which has the highest impact on CARE running time, but the same time, it affects the running time of the incremental algorithm only in a marginal way. Moreover, the size of the result set of a previous (more general) query usually is not an interesting parameter for mining algorithms although it is probably the most important one for the incremental algorithm.

The two algorithms have been assessed on the `purchases` database we introduced in the previous section. Some preliminary considerations on the influence that the typology of the dataset has on the incremental algorithms are necessary.

`purchases` is a sparse dataset. In sparse data, roughly speaking, the volume of results of a mining query, compared with the volume of the original database, is reduced and lower with respect to dense datasets at equal conditions (such as support threshold and constraints selectivity). We believe the main results on the behavior of the incremental algorithm we present in this Section with experiments on a sparse dataset should be still valid on a dense dataset with some differences. First, on a dense dataset the impact of I/O operations for reading the source database is less important if compared with the I/O required for reading the previous result set. This would constitute a disadvantage for the incremental algorithm. Second, in a dense dataset, the larger previous result set would allow a minor pruning on the search space which, on the other side, would be larger because data in a dense dataset are much more correlated. As a conclusion, the volume of the previous result set and that one of the search space should be two issues that should counterbalance each other. However, more insights on dense datasets are reserved for further work.

Extensive preliminary results on `purchases` showed that the incremental algorithm is substantially faster than CARE when a reduced previous result set is available. Indeed, this latter one allows incremental algorithms to do much pruning on the itemsets search space which finally results in a decisive advantage for the incremental algorithm. Hence, here, we want to test the incremental algorithms in a more stressing situation. We assumed that we have a single previous query available, which contained a very low support threshold (namely: 0.0085) and very loose predicates. As a result, the incremental algorithms have at their disposal in the BHF data structures a previous, very large (and thus not filtering) result set, composed of 158,336 rules. In the experiments, the current query is representative of a typical business-value scenario with a medium volume (6056 rules): find all the rules which appeared in 1.2 percent of the transactions and that contained costly to average priced items ("price ≥ 1000").

In the first pool of experiments, we varied just the size of the previous result set. We started from a previous result set which contained 7,931 rules (i.e., 5% of the total), and increased this number repeatedly until the whole volume of the 158,336 rules is reached. Figure 4 reports the result of the experiments. The number (N) of rules contained in the previous result set is plotted against the total running time of the two algorithms (y-axis). Here, we stress the fact that the two algorithms solve the same problem, but

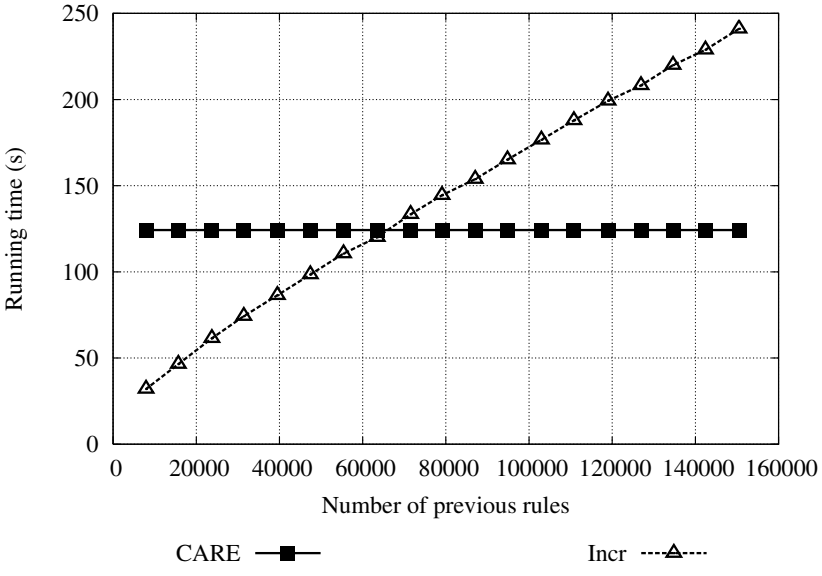


Fig. 4. CARE vs Incremental algorithm, as the cardinality of the previous query result set varies

while the incremental algorithm exploits the result of a previous query, CARE starts from scratch. As we can see, the running time of the incremental algorithm is very low in correspondence to low values of N (the number of rules in the previous result set). Then it increases linearly with N. We want to remark here this linear behavior is desirable, i.e., the scalability of incremental algorithms with respect to one of the main dimensions of the problem. On the other side, since the running time of CARE does not change with the tested parameter, the time spent by the incremental algorithm to solve the problem is bound to overcome the running time of CARE in the limit (i.e., when only a large volume of the previous result is available). In the experiment, this happens for $N = 65,998$ (accordingly to the line which interpolates the displayed data), but it should be noted that this value highly depends on the running time of CARE, i.e., on the support value and on the constraints of the current query. One may wonder why the incremental algorithm does not succeed in showing a better behavior than CARE in all situations (since the incremental algorithm has at its disposal more information than CARE). The answer is that the incremental algorithm has been thought and optimized in order to be extremely fast whenever a good (i.e., restricted and thus filtering) previous answer could be found in the system memory. In order to achieve the desired filtering of the search space, it mainly enforces the initial pruning provided by the results it reads from the disk. This, on one side, allows the algorithm to benefit from a very fast support counters update schema (and allows also a single pass over the database). On the other side, however, it forces the algorithm to read from disk another complete source of information that reveals a choice less competitive when its volume is large (and the pruning not sufficient). In conclusion, the incremental algorithm is a desirable strategy when a reduced previous result is available (if compared with the search space).

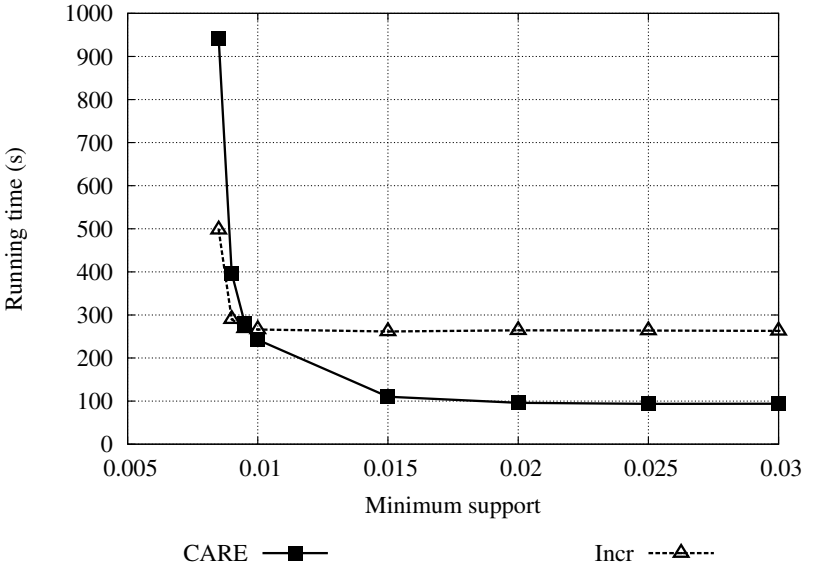


Fig. 5. CARE vs Incremental algorithm, as the support threshold of the current query varies

The trade off between the two algorithms is made evident once again by the results reported in the second set of experiments, shown in Figure 5, where the support threshold (x-axis) is plotted against the running time of the algorithms (y-axis). The previous result set contains the total volume of the large, above mentioned, previous result (158,336 rules). The mining constraints in current query are the same as the ones in the previous set of experiments but the support threshold in the current query varies between 0.0085 (the value set in the adopted, previous query) and 0.03. As it was expected, the incremental algorithm is much less affected by changes of the support threshold than CARE. In particular, we can see that its running time drops almost immediately to about 260 seconds and then it does not change very much. The reason for the drop in the execution time is that as the support threshold decreases, the number of rules given as output increases. Hence, it turns out that, in the current settings, the algorithm takes about 260 seconds to update the 158,336 support counters in BHF structure, while the rest of the time is spent in saving the result on the database. On the contrary, CARE by exploiting the antimonotonicity property of the constraints on minimum support, outperforms the incremental algorithm as the support threshold becomes high enough (reaching the lower bound of 100 seconds). Notably, according to the results reported in Figure 4, the chances are that if the previous query had contained less than 50,000 rules (instead of the $158,336$ present in the current setting), then the incremental algorithm would have outperformed CARE (with execution times lower than 100) no matter the value of the support threshold!

In the last pool of experiments, we set the support threshold equal to 0.0085, the previous results contain 158,336 rules, and let the selectivity of the constraints vary from 0.9010 to 0.9849. Figure 6 reports the results. As usual, the y-axis reports the total

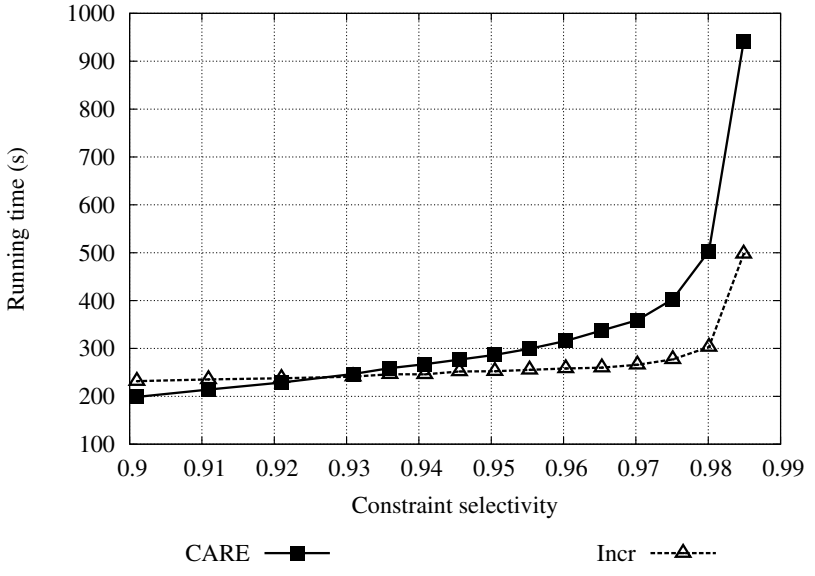


Fig. 6. CARE vs Incremental algorithm, as the constraints in the current query vary

running time of the algorithms, while the x-axis reports the percentage of the source table selected by the mining constraints (i.e., it reports the ratio between the number of rows that satisfy the constraints and the cardinality of the source table). As we can see the running time of both algorithms increases with the percentage of selected rows. Interestingly, this result suggests that the incremental algorithm, despite being faster on larger datasets, does not run as fast when the size of the database become smaller. This is probably due to the overhead the incremental algorithm suffers in order to build the BHF data structure. In order to check this hypothesis, we plotted in Figure 7 the total time needed by the algorithms in order to build the result once the preprocessing steps were completed[2]. As it can be seen, the "core" operations are consistently cheaper in the case of the incremental algorithm (this is expected, since it avoids the costly operations needed to manage the gid lists which are necessary in case of context dependent constraints).

In summary, the use of the incremental algorithm, is very often a winning choice when a suitable past result can be found. However, the choice between an incremental

[2] Which steps of each algorithm contribute to the preprocessing is hard to be stated objectively since the two algorithms make rather different choices in their early steps. However, we decided to consider as preprocessing the steps that are performed before the exploration of the itemsets search space occurs, which is usually a typical operation of the core data mining algorithm. In the case of CARE, we considered the time spent in reading the database as a preprocessing step since it is needed to fill the data structures used later on. In the case of the incremental algorithms we considered as pre-processing the time spent in building the BHF data structure, but not the time spent reading the DB since this one is interleaved with the "navigation" and management of the search space.

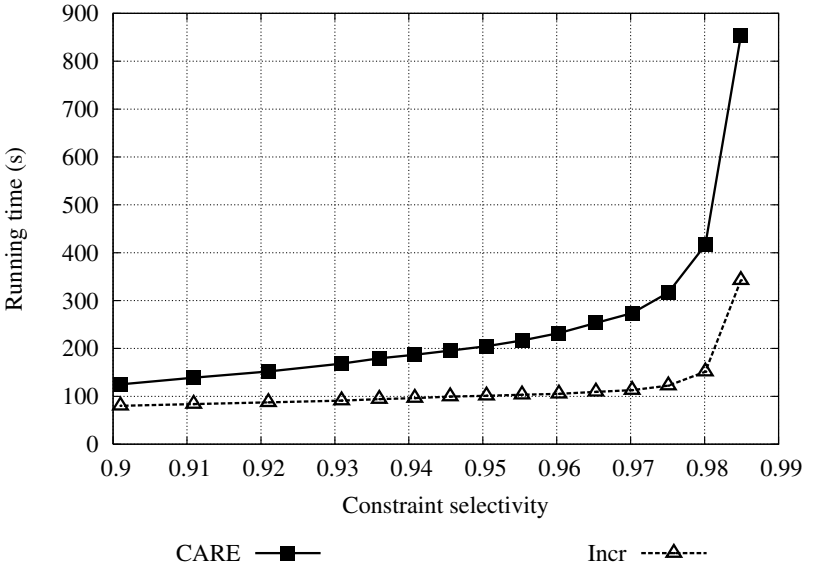


Fig. 7. CARE vs Incremental algorithm, as the constraints of the current query vary (times reported without the preprocessing time)

strategy and a non-incremental one should be made taking into account how the change of the problem dimensions affect the two strategies. In particular, it seems to be clear that whenever a very small previous result can be found, the incremental algorithm is hardly outperformed: it searches a small space and it builds the information needed to find the rules very efficiently. However, when the size of the previous result set grows larger, a "traditional" miner may win, especially when the support threshold is high. In this case, in fact, one looses both the advantages of the incremental algorithm: the algorithm will spend a large part of the time in building the BHF structure out of the previous result, and will probably search a larger space w.r.t. the space searched by algorithms which exploit the antimonotonicity of support.

8 Conclusions

In this paper we proposed a novel "incremental" approach to constraint-based mining which makes use of the information contained in previous results to answer new queries. The beneficial factors of the approach are that it uses both the previous results and the mining constraints, in order to reduce the itemsets search space.

We presented two incremental algorithms. The first one deals with item dependent constraints, while the second one with context dependent constraints. We note how the latter kind of constraints has been identified only recently and that there is very little support for them in current mining algorithms. However, the difficulty to solve mining queries with context dependent constraints can be partially overcome by combining

the "traditional" algorithms proposed so far in the literature, and the context dependent incremental algorithm proposed in this paper.

Moreover, we described a non-incremental algorithm (CARE) for the extraction of constrained association rules, in order to provide a direct comparison for the incremental ones. CARE is specifically designed to deal with context dependent constraints on both the body and the head of association rules and is, to the best of our knowledge, the only one of this type.

In Section 5 and in Section 7, we evaluated the incremental algorithms on a pretty large dataset. The results show that the approach reduces drastically the overall execution time. Whenever we have a small previous result to exploit, or when the support threshold is small, we believe the improvement is absolutely necessary in many practical data mining applications, in data warehouses and inductive database systems.

References

1. Agrawal, R., Imielinski, T., Swami, A.: Mining association rules between sets of items in large databases. In: Proc.ACM SIGMOD Conference on Management of Data, Washington, D.C., British Columbia (1993) 207–216
2. Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A.I.: Fast discovery of association rules. In Fayyad, U.M., Piatetsky-Shapiro, G., Smyth, P., Uthurusamy, R., eds.: Knowledge Discovery in Databases. Volume 2. AAAI/MIT Press, Santiago, Chile (1995)
3. Srikant, R., Vu, Q., Agrawal, R.: Mining association rules with item constraints. In: Proceedings of 1997 ACM KDD. (1997) 67–73
4. Ng, R.T., Lakshmanan, L.V.S., Han, J., Pang, A.: Exploratory mining and pruning optimizations of constrained associations rules. In: Proc. of 1998 ACM SIGMOD Int. Conf. Management of Data. (1998) 13–24
5. Tsur, D., Ullman, J.D., Abiteboul, S., Clifton, C., Motwani, R., Nestorov, S., Rosenthal, A.: Query flocks: A generalization of association-rule mining. In: Proceedings of 1998 ACM SIGMOD Int. Conf. Management of Data. (1998)
6. Chaudhuri, S., Narasayya, V., Sarawagi, S.: Efficient evaluation of queries with mining predicates. In: Proc. of the 18th Int'l Conference on Data Engineering (ICDE), San Jose, USA (2002)
7. Imielinski, T., Virmani, A., Abdoulghani, A.: Datamine: Application programming interface and query language for database mining. KDD-96 (1996) 256–260
8. Meo, R., Psaila, G., Ceri, S.: A new SQL-like operator for mining association rules. In: Proceedings of the 22st VLDB Conference, Bombay, India (1996)
9. Han, J., Fu, Y., Wang, W., Koperski, K., Zaiane, O.: DMQL: A data mining query language for relational databases. In Proc. of SIGMOD-96 Workshop on Research Issues on Data Mining and Knowledge Discovery (1996)
10. Wang, H., Zaniolo, C.: User defined aggregates for logical data languages. In: Proc. of DDLP. (1998) 85–97
11. Perng, C.S., Wang, H., Ma, S., Hellerstein, J.L.: Discovery in multi-attribute data with user-defined constraints. ACM SIGKDD Explorations **4** (2002) 56–64
12. Imielinski, T., Mannila, H.: A database perspective on knowledge discovery. Communications of the ACM **39** (1996) 58–64
13. Fang, M., Shivakumar, N., Garcia-Molina, H., Motwani, R., Ullman, J.: Computing iceberg queries efficiently. In: Proceeding of VLDB '98. (1998)
14. Sarawagi, S.: User-adaptive exploration of multidimensional data. In: Proc. of the 26th Int'l Conference on Very Large Databases (VLDB), Cairo, Egypt (2000) 307–316

15. Jeudy, B., Boulicaut, J.F.: Optimization of association rule mining queries. Intelligent Data Analysis **6** (2002) 341–357
16. Tuzhilin, A., Liu, B.: Querying multiple sets of discovered rules. In: KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining. (2002)
17. Baralis, E., Psaila, G.: Incremental refinement of mining queries. In: Proc. of First International Conference on Data Warehousing and Knowledge Discovery. Volume 1676 of Lecture Notes in Computer Science., Springer (1999) 173–182
18. Cheung, D.W., Han, J., Ng, V.T., Wong, C.Y.: Maintenance of discovered association rules in large databases: an incremental updating technique. In: ICDE96 12th International Conference on Data Engineering, New Orleans, Louisiana, USA (1996)
19. Lee, S.D., Cheung, D., Kao, B.: A general incremental technique for maintaining discovered association rules. In: Proceedings of the 5th International Conference On Database Systems For Advanced Applications, Melbourne, Australia (1997) 185–194
20. Thomas, S., Bodagala, S., Alsabti, K., Ranka, S.: An efficient algorithm for the incremental updation of association rules in large databases. In: KDD. (1997) 263–266
21. Labio, W., Yang, J., Cui, Y., Garcia-Molina, H., Widom, J.: Performance issues in incremental warehouse maintenance. In: Proceedings of Twenty-Sixth International Conference on Very Large Data Bases. (2000) 461–472
22. Meo, R., Botta, M., Esposito, R.: Query rewriting in itemset mining. In: Proceedings of the 6th International Conference On Flexible Query Answeringd Systems. LNAI 3055, Springer (2004)
23. Leung, C.K.S., Lakshmanan, L.V.S., Ng, R.T.: Exploiting succinct constraints using fp-trees. ACM SIGKDD Explorations **4** (2002) 40–49
24. Lu, H., Feng, L., Han, J.: Beyond intratransaction association analysis: mining multidimensional intertransaction association rules. ACM Trans. Inf. Syst. **18** (2000) 423–454
25. Feng, L., Dillon, T.S., Liu, J.: Inter-transactional association rules for multi-dimensional contexts for prediction and their application to studying meteorological data. Data Knowledge Engineering **37** (2001) 85–115
26. Grahne, G., Lakshmanan, L.V.S., Wang, X., Xie, M.H.: On dual mining: From patterns to circumstances, and back. In: Proceedings of the 17th International Conference on Data Engineering. (2001)
27. Bucila, C., Gehrke, J., Kifer, D., White, W.M.: Dualminer: a dual-pruning algorithm for itemsets with constraints. In: Proceedings of 2002 ACM KDD. (2002) 42–51
28. Bayardo, R., Agrawal, R., Gunopulos, D.: Constraint-based rule mining in large, dense databases. In: Proceedings of the 15th Int'l Conf. on Data Engineering, Sydney, Australia (1999)
29. Lakshmanan, L.V.S., Ng, R., Han, J., Pang, A.: Optimization of constrained frequent set queries with 2-variable constraints. In: Proceedings of 1999 ACM SIGMOD Int. Conf. Management of Data. (1999) 157–168
30. Raedt, L.D.: A perspective on inductive databases. ACM SIGKDD Explorations **4** (2002) 69–77
31. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings of the 20th VLDB Conference, Santiago, Chile (1994)
32. Savasere, A., Omiecinski, E., Navathe, S.: An efficient algorithm for mining association rules in large databases. In: Proceedings of the 21st VLDB Conference, Zurich, Switzerland (1995)